

# Finite Difference Stencils Implemented Using Chapel<sup>\*</sup>

Richard F. Barrett, Philip C. Roth, and Stephen W. Poole

Future Technologies Group  
Computer Science and Mathematics Division  
Oak Ridge National Laboratory  
Oak Ridge, TN 37831  
`rbarrett, rothpc, spoole@ornl.gov`  
<http://www.csm.ornl.gov/ft>

**Abstract.** Difference stencils are fundamental computations found throughout a broad range of scientific and engineering computer programs. In this report we present our experiences implementing stencils using Chapel, a global view programming language emerging from Cray as part of the DARPA High Productivity Computing Systems (HPCS) program. We found that Chapel allows us to express stencils with easy-to-write, readable, maintainable code, which significantly reduces the chance of programming errors. Furthermore, because the Chapel constructs we used represent high-level operations such as a reduction over a multi-dimensional array, more semantic information is provided to the compiler giving it more flexibility for producing efficient code for a given target platform. Although the current pre-release version of the Chapel compiler does not yet allow us to generate runtime performance statistics of our implementations, we discuss how a Chapel compiler might process the code in order to exploit architectural features, especially those of peta-scale parallel computing systems.

**Key words:** Scientific applications, finite difference methods, parallel programming.

## 1 Introduction

A broad range of physical phenomena in science and engineering can be described mathematically using partial differential equations. Determining the solution of these equations on computers is often accomplished using finite differencing methods. The algorithmic structure of these method maps naturally to the data parallel programming model.

Implementing these algorithms effectively is becoming increasingly difficult as high performance computing architectures become more complex. In order

---

<sup>\*</sup> This research was performed at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

to address this complexity, the DARPA High Productivity Computing Systems program[7] is funding the development of new programming languages: Chapel[4] and X10[12]. DARPA funded the initiation of the Fortress language[1], whose development continues.

Chapel is being developed as part of the Cray Cascade project. Like Fortress and X10, it is being designed to provide scientific application developers a means for easily expressing their algorithms in a form that also enables efficient computation on parallel distributed memory computing systems.

In this report, we present our experiences exploring the capabilities and expressiveness of Chapel, purposefully investigating syntactic and semantic approaches for implementing these difference stencils. We discuss how a Chapel compiler might translate our Chapel expressions of finite differencing methods to make effective use of the target platform’s characteristics. Although we are not presenting a true productivity study, we will point out issues relating to this complex notion as they are encountered.

We begin by describing the use of stencils in finite difference methods. Next, we present an overview of Chapel, and describe the syntax and semantics we used for this work. Then, we present a set of constructs designed to progressively lead to an effective implementation of these stencils from a single processor perspective. Following this, we discuss the constructs that distribute the problem across the parallel processes, then speculate on the performance issues associated with our implementations. Last, we offer our conclusions and present potential directions for our future research with Chapel.

## 2 Finite Difference Stencils

Finite difference methods are mathematical techniques for approximating derivatives or a differential operator by replacing the derivatives with some linear combination of discrete function values. An example of one such differential equation is Poisson’s equation:

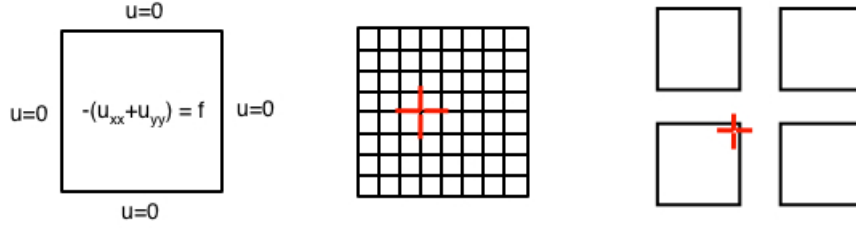
$$-(u_{xx} + u_{yy}) = f(x, y), \quad (1)$$

perhaps defined on

$$\Omega = [0, N] \times [0, N], \text{ with } u = 0 \text{ on } \delta\Omega.$$

(This equation is often written as  $\nabla^2\varphi = f$ .) We discretize  $\Omega$  with resolution  $1/h$ , resulting in  $(N/h) + 1 = n$  grid points in each dimension. During an iteration, each grid point is updated as a function of the current value of it and some combination of its neighbors. This computation is often described as applying a *stencil* to each point of the grid. We illustrate a 5-point stencil in two dimensions pictorially in Figure 1 and notationally as

$$u_{i,j}^{t+1} = \frac{u_{i,j-1}^t + u_{i-1,j}^t + u_{i,j}^t + u_{i+1,j}^t + u_{i,j+1}^t}{5}, \text{ for } i, j \in 1, \dots, n^2. \quad (2)$$



**Fig. 1.** Solving the 2d Poisson Equation using a 5-point difference stencil. The figure on the left shows the Poisson Equation defined on a continuous domain, with Dirichlet boundary conditions. The center figure shows the domain discretized, with a 5-point difference stencil, in red. The figure on the right shows the domain divided up as blocks for mapping to a parallel processing computer. Here the stencil requires data located on different parallel processes.

Figure 2 shows a parallel implementation of equation (2), using Fortran and some method of explicit parallelism for sharing the internal boundaries created by the decomposed domain, abstracted into procedure `EXCHANGE_BOUNDARY`. In a message passing environment, this abstraction would require the determination of neighbors with which to exchange messages, the location of the data to be transmitted, and the mechanics of the data transfer.

As described, our use of stencils presumes regular, equally spaced grid points across the global domain. This assumption significantly simplifies the implementation of the stencil algorithms, allowing us to focus on the aspects of interest in our experiments. However, an endless variety of stencils may be created. For example, excluding the points diagonally adjacent to the center point  $u_{i,j}$  creates

---

```

real, dimension(NROWS_LOCAL+2,NCOLS_LOCAL+2) :: GRID_NEW, GRID_OLD

call EXCHANGE_BOUNDARY ( ... )

do J = 2,NCOLS_LOCAL-1
  do I = 2,NROWS_LOCAL-1

    GRID_NEW(1,J) = (
      GRID_OLD(I-1,J) +
      GRID_OLD(I,J-1) + GRID_OLD(I,J) + GRID_OLD(I,J+1) +
      GRID_OLD(I+1,J) )
    / 5.0

  end do
end do

```

---

**Fig. 2.** Fortran implementation of a 5-point stencil in 2d.

a five-point stencil, dividing the sum by five rather than nine. In three dimensions, the five-point stencil may be extended to a seven-point stencil by including the neighbors to the front and rear of center point  $u_{i,j,k}$  and dividing by seven. Stencils may involve more, or even blocks of, distant neighbors. Furthermore, the contribution from a grid point may be quantified as a function of some weighting scheme other than the uniformity we presume here.

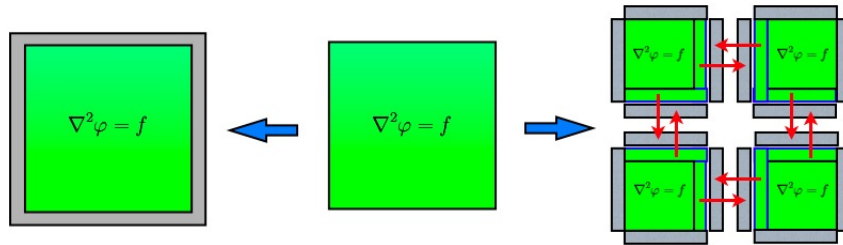
In the next section we briefly describe Chapel syntax and semantics, which will then allow us to present implementations of some of these stencils as well as discuss how others may be formulated.

### 3 Overview of Chapel

Scientific applications are most often parallelized using functionality defined by the Message Passing Interface (MPI)[18, 11]. Programs written using MPI define a collection of processes, each with its own private address space, which requires the code developer to explicitly manage the distribution and movement of data among the parallel processes. Popular alternatives include Co-array Fortran (CAF)[15], which makes data globally accessible via the co-array load and store semantics; Unified Parallel C (UPC)[10], which extends the C programming language to include a shared memory view of computation; and OpenMP[6], which presents a shared memory view of computation.

Chapel, as well as Fortress and X10, abstract the parallel processes from the view of the programmer. This “global view” model is designed to simplify the expression of a given algorithm as a parallel computation. This distinction is described with an example, extended from that described in section 2.

As graphically illustrated in Figure 3, a partial differential equation (here



**Fig. 3.** Global view vs. fragmented view parallel programming model.

again Poisson’s Equation) is defined on a two dimensional domain, illustrated by the center picture. Typically a rectangular domain is divided into blocks, with each block “assigned” to a parallel process. The stencil computation, then, induces an inter-process data sharing requirement along the data boundaries on a given process.

The fragmented view configuration for applying a solution algorithm on a parallel processing computer is shown on the right. Here the code developer must manage the interaction of the parallel processes as well as the overall data layout, including explicit control over the sharing of data among the individual blocks. This is usually accomplished by surrounding each block with a “halo” in order to control data movement (as indicated by the arrows) and maintain coherency. A global view language such as Chapel captures data associated with the problem in a single structure which it (as well as a fragmented model) may then surround with space for the physical boundary conditions. Although the code developer is not responsible for distributing and sharing data amongst the parallel processes, effective performance compels the language to provide a means for conveying information regarding parallelism in the problem.

Previous attempts at providing a global view of parallel computation have met with limited acceptance. With regard to HPF[16], from a performance perspective, this is attributable in large part to the lack of language constructs for effectively expressing the intent of the computation in a manner that could be exploited by the compiler and runtime system. And while OpenMP[6] presents a global view through the definition of a set of compiler directives and associated syntax, it is limited to regions of physically shared memory. In order to span multiple regions, OpenMP may be combined with MPI, which then creates a fragmented view.

Chapel, as well as Fortress and X10, strive to combine the strengths of these existing programming models while avoiding their weaknesses. Chapel pursues this goal by providing a global view of parallel programming based around the definition of a *domain*. It gives the programmer control over the parallelism of the application, specifically the data distribution and associated inter-process data sharing. The overall goal is to combine a global view of the program with the tools necessary for injecting high-level programmer “intent” that the compiler cannot easily discover in more traditional programming models. With its global programming model, Chapel bears some similarities to HPF, ZPL[3], and the Cray MTA extensions to C and Fortran[5].

At the time of this writing, the Chapel language specification is at version 0.702. A prototype compiler (pre-release version 0.4) has been provided to a small group of programmers who are gaining experience and providing feedback to the Chapel developers. However, the compiler is limited to serial “proof-of-concept” execution, and thus performance data will not be reported herein.

## Syntax

Unlike languages like HPF, CAF, UPC, and Titanium, which extend well-established serial languages to include a parallel processing capability, Chapel defines a new language. However, code developers will likely find its semantics and syntax familiar in many ways. Like C, executable statements are terminated by a semi-colon, brackets define executable blocks, variables can be declared (and initialized) within executable blocks, and variables can be re-cast. Like Fortran, multidimensional array indices, bounded by user defined values, are indexed

within parenthesis, variables can be explicitly typed or defined by their initialized value, and modules define name spaces and are included in the compilation unit via the `use` statement. Like C#, C++, and Java, object-oriented programming is possible with Chapel, and comments can either be captured within a block by slash-star star-slash (`/* Comment */`) or prepended by double slash (`//`).

Chapel lets the code developer define a broad set of data structures using a *domain*. In some (limited) sense, this is analogous to Fortran, in that it can be viewed as defining the size and shape of an array. However, domains are not limited to rectilinear structures: they can, for example, describe sparse arrays, graphs, sequences, and indexing sets, making Chapel applicable to a broad range of scientific applications.

Multi-dimensions may be described using *tuples* which aid readability by combining index counters (*i* and *j*) into a single variable *k*. Its important to note that Chapel does not impose a memory layout requirement on the arrays defined using these data structures.

### Parallel Semantics

Chapel provides a global view of parallelism by qualifying a domain with a *distribution*, which defines how arrays allocated with those domains are decomposed across the parallel processes. These mechanisms enable a parallel perspective of the computation regardless of the underlying mechanisms for parallel execution. Current distribution options include block, cyclic, block-cyclic, and cut.

The difference stencils described in this report were written using the data parallel model, based on *arithmetic domains*, which are rectilinear sets of Cartesian indices of an arbitrary rank, and *sparse domains*, which are subset of arithmetic domains. The `forall` iterator is the mechanism for expressing a parallel computation over a domain's indices. The combination of the domain, distribution, the `forall` iterator, tuples, and the unconstrained memory model provides the compiler and runtime system with significant flexibility for mapping the problem space to the architecture.

Task parallelism is supported, though not used in our work reported herein.

With regard to parallel programming, Chapel bears some similarities to HPF, ZPL[3], and the Cray MTA extensions to C and Fortran[5].

## 4 Implementations

We begin by defining a 9-point stencil in two dimensions, which includes the grid points diagonally adjacent to the center point in the 5-point stencil described in Equation 2. We extend this implementation to three dimensions by defining a 27-point stencil. We then modify these stencils by applying weights to the grid points. Next, using sparse domains, we construct 5-point and 7-point stencils in two and three dimensions, respectively. We then define the domains using distributions, so that the data and associated operations are spread across the

parallel processes. Finally, we highlight Chapel’s polymorphic capability for these computations.

#### 4.1 Basic stencils

The first task in implementing a function that applies a difference stencil is to define Chapel domains over which to iterate. We define an arithmetic domain `ProblemSpace` which describes the grid points in the physical domain. We define a second domain, `AllSpace` which describes the grid points plus “ghost points” for applying the boundary conditions<sup>1</sup>. Our first cut at a Chapel implementation of the nine-point stencil is shown in Figure 4.

---

```

const
  PhysicalSpace = [1..m, 1..n],      // Grid points in the 2d physical domain.
  AllSpace = PhysicalDomain.expand(1); // Physical domain plus boundary.

var
  newGrid, oldGrid
    : [AllSpace] real;

// Define neighbors:

const
  NW=(-1,-1), N=(-1,0), NE=(-1,1), W=(0,-1), E=(0,1), SW=(1,-1), S=(1, 0), SE=(1,1);

forall k in PhysicalSpace do

  newGrid(k) = (
    oldGrid(k+NW) + oldGrid(k+N) + oldGrid(k+NE) +
    oldGrid(k+W) + oldGrid(k) + oldGrid(k+E) +
    oldGrid(k+SW) + oldGrid(k+S) + oldGrid(k+SE) )
    / 9.0;

```

---

**Fig. 4.** Chapel 9-point stencil on a 2d domain using tuple arithmetic.

Array offset indexing complexity is reduced and readability is increased by combining the `i` and `j` indices into the single tuple `k` and defining parameterized tuple addition<sup>2</sup>. In addition to coding simplification, the global view of the iteration, enhanced by the single loop expression enabled by the use of the tuple `k`, provides greater flexibility to the compiler for applying its strategies for achieving performance. For example, unconstrained by a requirement for row- or column-major ordering, the compiler may lay out memory as it sees fit. Moreover, data may be decomposed across the parallel processes in a manner that maps best to the particular architecture, taking advantage of memory hierarchies, processor heterogeneity, and perhaps other features of the computing environment.

<sup>1</sup> A new feature lets us define `AllSpace` as an *expansion* of `ProblemSpace`. The syntax is `AllSpace = ProblemSpace.expand(1)`. In addition to coding convenience, this relationship could be exploited by the compiler.

<sup>2</sup> Tuple addition is explicitly defined in our code, but is slated for inclusion in the Chapel language specification.

We can improve this implementation by noting that the application of the stencil operation to a grid point and its neighbors may be posed as a reduction. This perspective enables an even more compact and descriptive implementation than our initial attempt, as shown in Figure 5.

---

```

const
  PhysicalSpace = [1..m, 1..n],
  AllSpace = PhysicalDomain2d.expand(1),
  Stencil9pt = [-1..1, -1..1];

var
  newGrid, oldGrid
    : [AllSpace] real;

forall k in PhysicalSpace do
  newGrid(k) = ( + reduce [i in Stencil] oldGrid(k+i) ) / 9.0;

```

---

*Analogously, a 27-point stencil in a 3d domain could be written as*

---

```

const
  PhysicalSpace = [1..m, 1..n, 1..p],
  AllSpace = PhysicalDomain3d.expand(1),
  Stencil = [-1..1, -1..1, -1..1];

var
  newGrid, oldGrid
    : [AllSpace] real;

forall k in PhysicalSpace do
  newGrid(k) = ( + reduce [i in Stencil] oldGrid(k+i) ) / 27.0;

```

---

**Fig. 5.** Chapel reduction operator based stencils

The notation

`[ i in Stencil ]`

is equivalent to

`forall i in Stencil do`

which aids readability in this context.

In addition to producing more readable code, this reduction-based approach conveys the intent of the stencil computation to the compiler in addition to the data structure in the domain definition, possibly enabling the compiler to produce a more efficient computation for a given target platform.

## 4.2 Weighted stencils

In many contexts the stencil involves a weighting of the grid points. This may be necessary in order to impart an actual contribution of the grid point, or may

even be for other purposes, such as a formulation of a matrix-vector product, where the “weights” are actually the matrix coefficients.

Adding weights to the grid points in a Fortran stencil implementation, analogous to the 5-point stencil shown in Figure 2, increases the complexity of the code, especially with regard to indexing requirements, as illustrated in Figure 6.

---

```

real, dimension(NROWS_LOCAL+2,NCOLS_LOCAL+2) :: A, X, Y

do J = 2,NCOLS_LOCAL-1
  do I = 2,NROWS_LOCAL-1

    Y(I,J) =
      A(I-1,J-1)*X(I-1,J-1)+A(I-1,J)*X(I-1,J)+A(I-1,J+1)*X(I-1,J+1)+ &
      A(I,J-1)*X(I,J-1)+A(I,J)*X(I,J)+A(I,J+1)*X(I,J+1)+ &
      A(I+1,J-1)*X(I+1,J-1)+A(I+1,J)*X(I+1,J)+A(I+1,J+1)*X(I+1,J+1)
  end do
end do

```

---

**Fig. 6.** Fortran weighted stencils

Adding this same weighting scheme to the Chapel implementation shown in Figure 5 simply requires the insertion of the weighting array, as shown in Figure 7.

---

```

const
  PhysicalSpace = [1..m, 1..n, 1..p],
  AllSpace = PhysicalDomain.expand(1);
  Stencil = [-1..1, -1..1, -1..1];

var
  A, X, Y
  : [AllSpace] real;

forall k in PhysicalSpace do
  Y(k) = ( + reduce [i in Stencil] A(k+i) * X(k+i));

```

---

**Fig. 7.** Chapel reduction operator based weighted stencils

### 4.3 The 5-pt stencil: sparse domains

At first glance the 5-point stencil might be viewed as a subset of the 9-point stencil. While true from the Fortran+MPI perspective as well as for the Chapel implementation using parameterized tuple arithmetic (analogous to the implementation shown in Figure 4), it does not map directly to the reduction configuration we prefer since we can’t define the stencil as a regular block required by the arithmetic domain. Thus far we have two options for addressing this. First,

we could view the 5-point stencil as a 9-point stencil, setting corner coefficients to zero (with the associated multiplication perhaps recognized and eliminated by a compiler[9]). This has the advantage of simplicity, and could result in strong performance due to the regular blocks. However, we don't want to make such presumptions here, and more importantly, a language should be able to support this operation as well as it supports the 9-point stencil.

Our solution is to configure the stencil as a *sparse* domain, defined as a subset of the 9-point stencil arithmetic domain. While this capability is not yet implemented in the compiler, nor even fully specified, we can sketch the idea, shown in Figure 8.

---

```

1  const
2      PhysicalSpace = [1..m, 1..n]
3      AllSpace = PhysicalSpace.expand(1);

4      Stencil9pt = [-1..1, -1..1],
5      Stencil: sparse subdomain(Stencil9pt) = ( (-1,0), (0,-1), (0,0), (0,1), (1,0) );

6  var
7      A, X, Y
8      : [PhysicalSpace] real;

9  forall i in PhysicalSpace do
10     Y(i) = ( + reduce [k in Stencil] A(i+k)*X(i+k);

```

---

**Fig. 8.** Chapel 2d 5-point stencil.

The sparse domain creates the 5-point stencil by selecting a subset of the dense arithmetic domain which defines the 9-point stencil. As with the 9-point stencil, the reduction operator is controlled by the **Stencil** domain, providing access into the grid point data and their weights. The stencil pattern can be set several ways, including as a runtime conditional statement (shown in Figure 9), which might be useful in other situations.

---

```

1  const
2      Stencil: sparse subdomain(Stencil9pt) = [(i,j) in Stencil9pt]
3      if ( abs(i) + abs(j) < 2 ) then (i,j);

```

---

**Fig. 9.** Runtime conditional stencil configuration.

#### 4.4 Parallel implementation

Typically a rectangular domain is divided into blocks, with each block "assigned" to a parallel process. The stencil computation, then, induces an inter-process data sharing requirement along the data boundaries on a given process.

As previously discussed in section 3, when using the fragmented-view model, the code developer typically divides the domain into blocks, assigning each to a parallel process. For our purposes here, we will presume the same decomposition for the Chapel implementation. However, the code developer simply qualifies the domain definition with this decomposition and lets the compiler and runtime system take care of the details. Based on available hardware capabilities, we could envision a configuration that does not simply map this to a halo exchange algorithm[2]. However, a halo distribution has been proposed[8], which may be desirable in many situations.

An array is decomposed across the parallel processes by adding a distribution to the domain that defines its structure. The syntax for this is shown in Figure 10.

---

```
const
  PhysicalSpace : domain(2) distributed(Block) = [1..m, 1..n],
  AllSpace = PhysicalDomain.expand(1);

var
  newGrid, oldGrid
    : [AllSpace] real;
```

---

**Fig. 10.** Domains defined with a block distribution across the parallel processes. The red highlighted text is the distribution syntax.

#### 4.5 Polymorphism

The alert reader will notice an important characteristic of our reduction-based implementations: we can write the actual computation so that it need not change as we apply different stencils to different dimensions. That is, we can change the stencil simply by changing the definition of the array domain.

---

```
forall k in PhysicalSpace do
  Y(k) = ( + reduce [i in Stencil] A(k+i) * X(k+i)) / val;
```

---

**Fig. 11.** Polymorphic Chapel stencil operator

We qualify this observation with the realization that we may not want to include the weighting array or the divisor scalar for all situations. However, regardless of the sort of structure the domains define (e.g. rectilinear, sparse, graph, etc.), the computation code can remain unchanged. And since domains are defined as variables (or parameters), they may be passed throughout an application as could any other variable or parameter.

## 5 A Brief Note Regarding Performance Expectations

For our purposes, Chapel success is strongly tied to its performance on HPC platforms. In a sense then, our work here forms the basis for a study on the performance potential of Chapel.

The global expression of an algorithm can provide meaningful flexibility to the compiler, enabling a basis for strong use of asynchronous communication, sharing data as it is needed, taking advantage of architecture specific runtime capabilities, etc. For example, a reduction over an array whose structure is well-defined provides a strong description of the intent of the stencil computation. The fragmented view compels the use of halos, which injects synchronization in order to manage the transfer of data. Although we can envision a similar implementation by a Chapel compiler, this is not an inherent characteristic. Further, the lack of a memory layout constraint may enable even more effective use of architecture specific characteristics.

As Chapel compilers mature, We look forward to analyzing the performance of these stencil computations as well as our developing implementations of other computational kernels as compiler development progresses.

## 6 Conclusions and Future Work

In this report we described our experiences implementing difference stencils using Chapel, a language emerging from the Cray Cascade project as part of the DARPA HPCS program. With Chapel, we were able to write concise descriptions of multidimensional stencil code for use in, for example, solving discretized partial differential equations. This combination of computation and data structure specification will provide the compiler writer with the information necessary to produce a higher level of optimization for the application. This in turn will enable a greater level of performance for the system. Prior work in this area leads us to believe these stencil computations can execute at high efficiencies using this desirable global view programming model[17, 9]. When support for parallel computation becomes available, we will evaluate and report on the performance of the code presented in this report.

In addition to the work described in this paper, we are investigating the use of Chapel to express the ASC[14] computational kernel known as “sweep3d”[13]. This kernel forms the heart of application programs that solve the deterministic neutron transport problem. Current work is mostly a translation of the existing Fortran/MPI implementation, but we are also designing a Chapel implementation of this computation from First Principles.

In addition to Chapel, we are investigating the suitability of the new languages X-10 from IBM[12] and Fortress from Sun[1] to this and other work. Continued work in this direction will enable future comparisons of the strengths and weaknesses of these languages for parallel scientific computing.

**Acknowledgements:** We are grateful to Brad Chamberlain, the Technical Lead of the Chapel project, for invaluable discussions, advice, and code review. He has taken seriously our many discussions regarding our perceptions of the value of existing as well as potential functionality in the Chapel language. We extend this gratitude to all members of the Chapel team who, through Brad, aided our efforts.

## References

1. E. Allen, D. Chase, J. Hallet, V. Luchangco, J. Maessen, S. Ryu, G. L. Steele Jr, and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. $\beta$ . Technical report, Sun Microsystems, Inc., 2007.
2. R.F. Barrett. Co-Array Fortran experiences solving PDE using finite differencing schemes. In *Proceedings of the 48th Cray User Group*, May 2006.
3. B.L.Chamberlain. *The Design and Implementation of a Region-Based Parallel Programming Language*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 2001.
4. B.L. Chamberlain, D.Callahan, and H.P. Zima. Parallel programming and the Chapel language. *International Journal on High Performance Computer Applications*, To appear, 2007.
5. Cray, Inc. Cray MTA-2 Programmer’s Guide. S-2320-10, 2005.
6. L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 1998.
7. DARPA. High Productivity Computing Systems program. <http://www.darpa.mil/ipto/programs/hpcs>, 1999.
8. R.E. Diaconescu and H.P. Zima. An approach to data distribution in Chapel. *International Journal on High Performance Computer Applications*, To appear, 2007.
9. S. J. Dietz, B.L. Chamberlain, and L. Snyder. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.
10. T.A. El-Ghazawi, W.W. Carlson, and J.M. Draper. UPC language specification, version 1.1.1. <http://www.gwu.edu/~upc/documentation.html>.
11. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference: Volume 2 - The MPI-2 Extensions*. The MIT Press, 1998.
12. IBM. Report on the experimental language X10, draft v 0.41. IBM TJ Watson Research Center, [http://domino.research.ibm.com/comm/research\\_projects.nsf/pages/x10.index.html](http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html), February, 2006.
13. Accelerated Strategic Computing Initiative. The ASCI sweep3d Benchmark Code. <http://www.llnl.gov/asci/benchmarks/asci/limited/sweep3d>, 1995.
14. Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratories. The Advanced Simulation and Computing. <http://www.sandia.gov/NNSA/ASC>, 1995.
15. R.W. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998. <http://www.co-array.org>.
16. High Performance Fortran Forum. High Performance Fortran language specification. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993.

17. G. Roth, J. Mellor-Crumey, K. Kennedy, and R. G. Brickner. Compiling stencils in high performance fortran. In *SC'97: High Performance Networking and Computing*, 1997.
18. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition*. The MIT Press, 1998.